15 Using the platform features, part 1

This chapter covers

- Understanding the built-in features provided by ASP.NET Core
- Accessing the application configuration
- Storing secrets outside of the project folder
- Logging messages
- Generating static content and using client-side packages

ASP.NET Core includes a set of built-in services and middleware components that provide features that are commonly required by web applications. In this chapter, I describe three of the most important and widely used features: application configuration, logging, and serving static content. In chapter 16, I continue to describe the platform features, focusing on the more advanced built-in services and middleware. Table 15.1 puts the chapter in context.

Table 15.1 Putting platform features in context

Question	Answer
What are they?	The platform features deal with common web application requirements, such as configuration, logging, static files, sessions, authentication, and database access.
Why are they useful?	Using these features means you don't have to re-create their functionality in your own projects.
How are they used?	The built-in middleware components are added to the request pipeline using extension methods whose name starts with `Use`. Services are set up using methods that start with `Add`.
Are there any pitfalls or limitations?	The most common problems relate to the order in which middleware components are added to the request pipeline. Middleware components form a chain along which requests pass, as described in chapter 12.
Are there any alternatives?	You don't have to use any of the services or middleware components that ASP.NET Core provides.

Table 15.2 provides a guide to the chapter.

Table 15.2 Chapter guide

Problem	Solution	Listing
Accessing the configuration data	Use the `IConfiguration` service.	4–8
Setting the application environment	Use the launch settings file.	9–11
Determining the application environment	Use the `IWebHostEnvironment` service.	12
Keeping sensitive data outside of the project	Create user secrets.	13–17
Logging messages	Use the `ILogger<T>` service.	18–27
Delivering static content	Enable the static content middleware.	28–31
Delivering client-side packages	Install the package with LibMan and deliver it with the static content middleware.	32–35

15.1 Preparing for this chapter

In this chapter, I continue to use the Platform project created in chapter 14. To prepare for this chapter, update the `Program.cs` file to remove middleware and services, as shown in listing 15.1.

Listing 15.1 The contents of the Program.cs file in the Platform folder

```
var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.MapGet("/", async context => {
    await context.Response.WriteAsync("Hello World!");
});

app.Run();
```

One of the main topics in this chapter is configuration data. Replace the contents of the `appsettings.Development.json` file with the contents of listing 15.2 to remove the setting added in chapter 14.

Listing 15.2 The contents of the appsettings.Development.json file in the Platform folder

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    }
  }
}
```

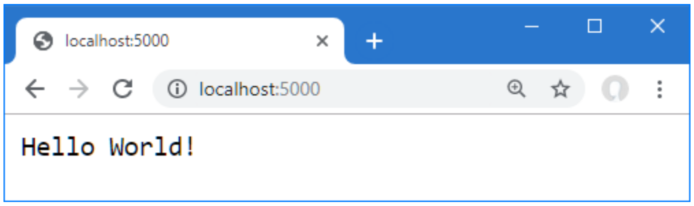
Start the application by opening a new PowerShell command prompt, navigating to the `Platform` project folder, and running the command shown in listing 15.3.

> **TIP** You can download the example project for this chapter—and for all the other chapters in this book—from https://github.com/manningbooks/pro-asp.net-core-7. See chapter 1 for how to get help if you have problems running the examples.

Listing 15.3 Starting the ASP.NET Core runtime

```
dotnet run
```

Open a new browser tab and navigate to http://localhost:5000; you will see the content shown in figure 15.1.



```
Hello World!
```

Figure 15.1 Running the example application

15.2 Using the configuration service

One of the built-in features provided by ASP.NET Core is access to the application's configuration settings, which is presented as a service.

The main source of configuration data is the `appsettings.json` file. The `appsettings.json` file created by the template used in chapter 12 contains the following settings:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

The configuration service will process the JSON configuration file and create nested configuration sections that contain individual settings. For the `appsettings.json` file in the example application, the configuration service will create a `Logging` configuration section that contains a `LogLevel` section. The `LogLevel` section will contain settings for `Default` and `Microsoft.AspnetCore`. There will also be an `AllowedHosts` setting that isn't part of a configuration section and whose value is an asterisk (the `*` character).

The configuration service doesn't understand the meaning of the configuration sections or settings in the `appsettings.json` file and is just responsible for processing the JSON data file and merging the configuration settings with the values obtained from other sources, such as environment variables or command-line arguments. The result is a hierarchical set of configuration properties, as shown in figure 15.2.

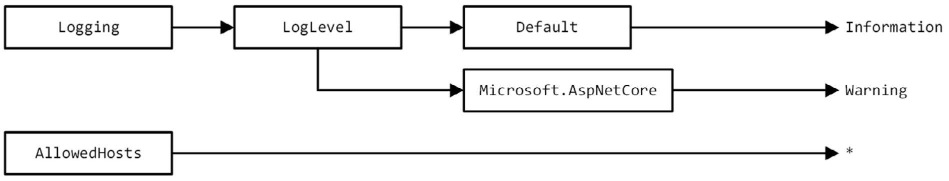


Figure 15.2 The hierarchy of configuration properties in the appsettings.json file

15.2.1 Understanding the environment configuration file

Most projects contain more than one JSON configuration file, allowing different settings to be defined for different parts of the development cycle. There are three predefined environments, named `Development`, `Staging`, and `Production`, each of which corresponds to a commonly used phase of development. During startup, the configuration service looks for a JSON file whose name includes the current environment. The default environment is `Development`, which means the configuration service will load the `appsettings.Development.json` file and use its contents to supplement the contents of the main `appsettings.json` file.

NOTE The Visual Studio Solution Explorer nests the `appsettings.Development.json` file in the `appsettings.json` item. You can expand the `appsettings.json` file to see and edit the nested entries or click the button at the top of the Solution Explorer that disables the nesting feature.

Here are the configuration settings added to the `appsettings.Development.json` file in listing 15.2:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    }
  }
}
```

Where the same setting is defined in both files, the value in the `appsettings.Development.json` file will replace the one in the `appsettings.json` file, which means that the contents of the two JSON files will produce the hierarchy of configuration settings shown in figure 15.3.

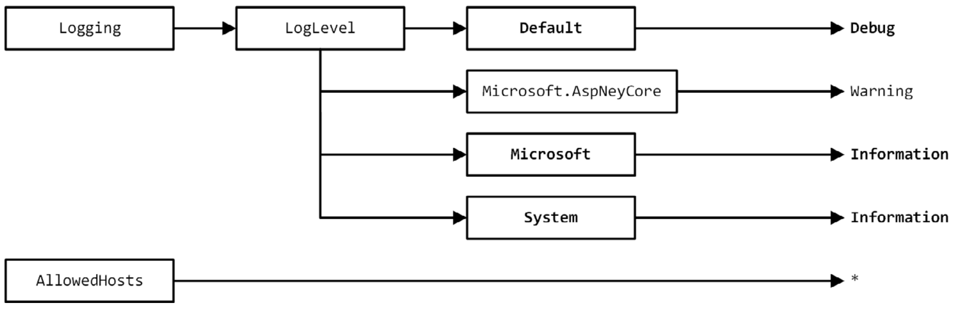


Figure 15.3 Merging JSON configuration settings

The effect of the additional configuration settings is to increase the detail level of logging messages, which I describe in more detail in the "Using the Logging Service" section.

15.2.2 Accessing configuration settings

The configuration data is accessed through a service. If you only require the configuration data to configure middleware, then the dependency on the configuration service can be declared using a parameter, as shown in listing 15.4.

Listing 15.4 Accessing configuration data in the Program.cs file in the Platform folder

```
var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.MapGet("config", async (HttpContext context,
        IConfiguration config) => {
    string? defaultDebug = config["Logging:LogLevel:Default"];
    await context.Response
        .WriteAsync($"The config setting is: {defaultDebug}");
});

app.MapGet("/", async context => {
    await context.Response.WriteAsync("Hello World!");
});

app.Run();
```

Configuration data is provided through the `IConfiguration` interface; this interface is defined in the `Microsoft.Extensions.Configuration` namespace and provides an API for navigating through the configuration hierarchy and reading configuration settings. Configuration settings can be read by specifying the path through the configuration sections, like this:

```
...
string? defaultDebug = config["Logging:LogLevel:Default"];
...
```

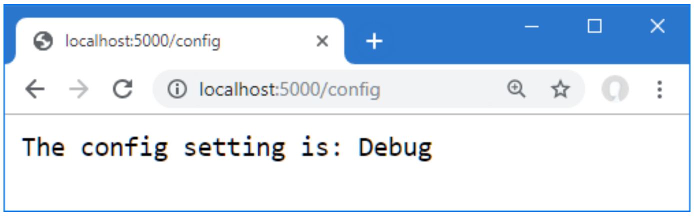
This statement reads the value of the `Default` setting, which is defined in the `LogLevel` section of the `Logging` part of the configuration. The names of the configuration sections and the configuration settings are separated by colons (the `:` character).

The value of the configuration setting read in listing 15.4 is used to provide a result for a middleware component that handles the `/config` URL. Restart ASP.NET Core using Control+C at the command prompt and run the command shown in listing 15.5 in the `Platform` folder.

Listing 15.5 Starting the ASP.NET Core platform

```
dotnet run
```

Once the runtime has restarted, navigate to the http://localhost:5000/config URL, and you will see the value of the configuration setting displayed in the browser tab, as shown in figure 15.4.



```
The config setting is: Debug
```

Figure 15.4 Reading configuration data

15.2.3 Using the configuration data in the Program.cs file

As noted in chapter 14, the `WebApplication` and `WebApplicationBuilder` classes provide a `Configuration` property that can be used to obtain an implementation of the `IConfiguration` interface, which is useful when using configuration data to configure an application's services. Listing 15.6 shows both uses of configuration data.

Listing 15.6 Configuring services and pipeline in the Program.cs file in the Platform folder

```
var builder = WebApplication.CreateBuilder(args);

var servicesConfig = builder.Configuration;
// - use configuration settings to set up services

var app = builder.Build();

var pipelineConfig = app.Configuration;
// - use configuration settings to set up pipeline

app.MapGet("config", async (HttpContext context,
        IConfiguration config) => {
    string? defaultDebug = config["Logging:LogLevel:Default"];
    await context.Response
        .WriteAsync($"The config setting is: {defaultDebug}");
});

app.MapGet("/", async context => {
    await context.Response.WriteAsync("Hello World!");
});

app.Run();
```

This may seem like an unnecessary step because there is so little code in the `Program.cs` file in this example application, which makes it obvious that the configuration service isn't replaced. It isn't always as obvious in a real project, where services can be defined in groups by methods defined outside of the `Program.cs` file, making it difficult to see if these methods alter the `IConfiguration` service.

15.2.4 Using configuration data with the options pattern

In chapter 12, I described the options pattern, which is a useful way to configure middleware components. A helpful feature provided by the `IConfiguration` service is the ability to create options directly from configuration data.

To prepare, add the configuration settings shown in listing 15.7 to the `appsettings.json` file.

Listing 15.7 Adding configuration data in the appsettings.json file in the Platform folder

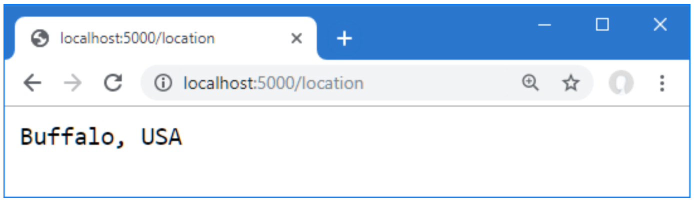
```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "Location": {
    "CityName": "Buffalo"
  }
}
```

The `Location` section of the configuration file can be used to provide options pattern values, as shown in listing 15.8.

Listing 15.8 Using configuration data in the Program.cs file in the Platform folder

```
using Platform;

var builder = WebApplication.CreateBuilder(args);

var servicesConfig = builder.Configuration;
builder.Services.Configure<MessageOptions>(
    servicesConfig.GetSection("Location"));

var app = builder.Build();

var pipelineConfig = app.Configuration;
// - use configuration settings to set up pipeline

app.UseMiddleware<LocationMiddleware>();

app.MapGet("config", async (HttpContext context,
        IConfiguration config) => {
    string? defaultDebug = config["Logging:LogLevel:Default"];
    await context.Response
        .WriteAsync($"The config setting is: {defaultDebug}");
});

app.MapGet("/", async context => {
    await context.Response.WriteAsync("Hello World!");
});

app.Run();
```

The configuration data is obtained using the `GetSection` method and passed to the `Configure` method when the options are created. The configuration values in the selected section are inspected and used to replace the default values with the same names in the options class. To see the effect, restart ASP.NET Core and use the browser to navigate to the http://localhost:5000/location URL. You will see the results shown in figure 15.5, where the `CityName` option is taken from the configuration data and the `CountryName` option is taken from the default value in the options class.



Buffalo, USA

Figure 15.5 Using configuration data in the options pattern

15.2.5 Understanding the launch settings file

The `launchSettings.json` file in the `Properties` folder contains the configuration settings for starting the ASP.NET Core platform, including the TCP ports that are used to listen for HTTP and HTTPS requests and the environment used to select the additional JSON configuration files.

> **TIP** Visual Studio often hides the `Properties` folder. If you can't see the folder, click the Show All Files button at the top of the Solution Explorer to reveal the folder and the `launchSettings.json` file.

Here is the content added to the `launchSettings.json` file when the project was created and then edited to set the HTTP ports:

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:5000",
      "sslPort": 0
    }
  },
  "profiles": {
    "Platform": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

The iisSettings section is used to configure the HTTP and HTTPS ports used when the ASP.NET Core platform is started through IIS Express, which is how older versions of ASP.NET Core were deployed.

The profiles section describes a series of launch profiles, which define configuration settings for different ways of running the application. The Platform section defines the configuration used by the dotnet run command. The IIS Express section defines the configuration used when the application is used with IIS Express.

Both profiles contain an environmentVariables section, which is used to define environment variables that are added to the application's configuration data. There is only one environment variable defined by default: ASPNETCORE_ENVIRONMENT.

During startup, the value of the ASPNETCORE_ENVIRONMENT setting is used to select the additional JSON configuration file so that a value of Development, for example, will cause the appsettings.Development.json file to be loaded.

When the application is started within Visual Studio Code, ASPNETCORE_ENVIRONMENT is set in a different file. Select Run > Open Configurations to open the launch.json file in the .vscode folder, which is created when a project is edited with Visual Studio Code. Here is the default configuration for the example project, showing the current ASPNETCORE_ENVIRONMENT value, with the comments removed for brevity:

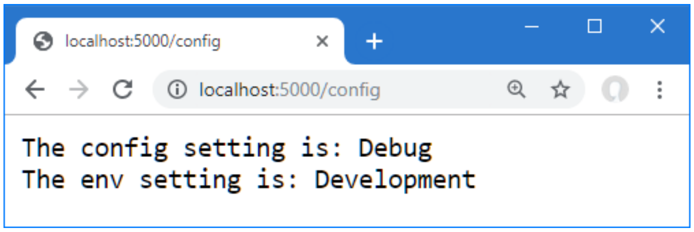
```
{
    "version": "0.2.0",
    "configurations": [
        {

            "name": ".NET Core Launch (web)",
            "type": "coreclr",
            "request": "launch",
            "preLaunchTask": "build",
            "program": "${workspaceFolder}/bin/Debug/net7.0/Platform.dll",
            "args": [],
            "cwd": "${workspaceFolder}",
            "stopAtEntry": false,
            "serverReadyAction": {
                "action": "openExternally",
                "pattern": "\\bNow listening on:\\s+(https?://\\S+)"
            },
            "env": {
                "ASPNETCORE_ENVIRONMENT": "Development"
            },
            "sourceFileMap": {
                "/Views": "${workspaceFolder}/Views"
            }
        },
        {
            "name": ".NET Core Attach",
            "type": "coreclr",
            "request": "attach"
        }
    ]
}
```

To display the value of the ASPNETCORE_ENVIRONMENT setting, add the statements to the middleware component that responds to the /config URL, as shown in listing 15.9.

Listing 15.9 Displaying the configuration in the Program.cs file in the Platform folder

```
using Platform;

var builder = WebApplication.CreateBuilder(args);

var servicesConfig = builder.Configuration;
builder.Services.Configure<MessageOptions>(
    servicesConfig.GetSection("Location"));

var app = builder.Build();

var pipelineConfig = app.Configuration;

app.UseMiddleware<LocationMiddleware>();

app.MapGet("config", async (HttpContext context,
        IConfiguration config) => {
    string? defaultDebug = config["Logging:LogLevel:Default"];
    await context.Response
        .WriteAsync($"The config setting is: {defaultDebug}");
    string? environ = config["ASPNETCORE_ENVIRONMENT"];
    await context.Response.WriteAsync($"\nThe env setting is: {environ}");
});

app.MapGet("/", async context => {
    await context.Response.WriteAsync("Hello World!");
});

app.Run();
```

Restart ASP.NET Core and navigate to http://localhost:5000/config, and you will see the value of the ASPNETCORE_ENVIRONMENT setting, as shown in figure 15.6.



```
The config setting is: Debug
The env setting is: Development
```

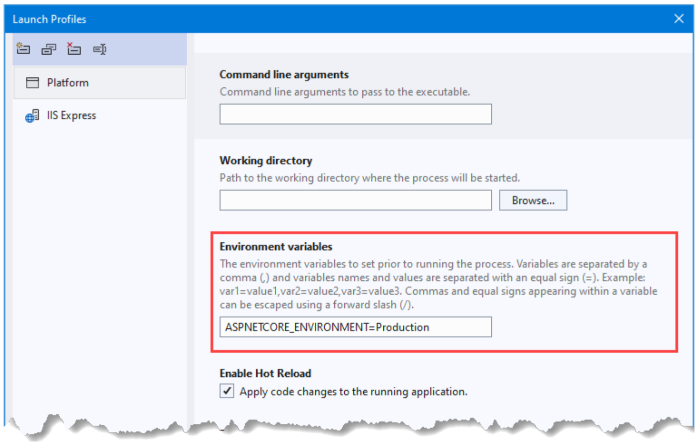
Figure 15.6 Displaying the environment configuration setting

To see the effect that the ASPNETCORE_ENVIRONMENT setting has on the overall configuration, change the value in the launchSettings.json file, as shown in listing 15.10.

Listing 15.10 Changing the launchSettings.json file in the Platform/Properties folder

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:5000",
      "sslPort": 0
    }
  },
  "profiles": {
    "Platform": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Production"
      }
    },
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

If you are using Visual Studio, you can change the environment variables by selecting Debug > Launch Profiles. The settings for each launch profile are displayed, and there is support for changing the value of the `ASPNETCORE_ENVIRONMENT` variable, as shown in figure 15.7.



Command line arguments
Command line arguments to pass to the executable.

Working directory
Path to the working directory where the process will be started.
Browse...

Environment variables
The environment variables to set prior to running the process. Variables are separated by a comma (,) and variables names and values are separated with an equal sign (=). Example: var1=value1,var2=value2,var3=value3. Commas and equal signs appearing within a variable can be escaped using a forward slash (/).

ASPNETCORE_ENVIRONMENT=Production

Enable Hot Reload
✓ Apply code changes to the running application.

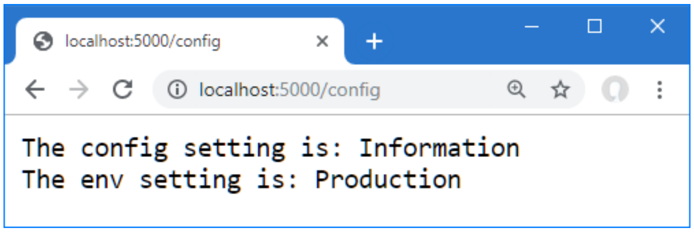
Figure 15.7 Changing an environment variable using Visual Studio

If you are using Visual Studio Code, select Run > Open Configurations and change the value in the env section, as shown in listing 15.11.

Listing 15.11 Changing the launch.json file in the Platform/.vscode folder

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": ".NET Core Launch (web)",
            "type": "coreclr",
            "request": "launch",
            "preLaunchTask": "build",
            "program": "${workspaceFolder}/bin/Debug/net7.0/Platform.dll",
            "args": [],
            "cwd": "${workspaceFolder}",
            "stopAtEntry": false,
            "serverReadyAction": {
                "action": "openExternally",
                "pattern": "\\bNow listening on:\\s+(https?://\\S+)"
            },
            "env": {
                "ASPNETCORE_ENVIRONMENT": "Production"
            },
            "sourceFileMap": {
                "/Views": "${workspaceFolder}/Views"
            }
        },
        {
            "name": ".NET Core Attach",
            "type": "coreclr",
            "request": "attach"
        }
    ]
}
```

Save the changes to the property page or configuration file and restart ASP.NET Core. Navigate to http://localhost:5000/config, and you will see the effect of the environment change, as shown in figure 15.8.



```
The config setting is: Information
The env setting is: Production
```

Figure 15.8 The effect of changing the environment configuration setting

Notice that both configuration values displayed in the browser have changed. The appsettings.Development.json file is no longer loaded, and there is no appsettings.Production.json file in the project, so only the configuration settings in the appsettings.json file are used.

15.2.6 Using the environment service

The ASP.NET Core platform provides the IWebHostEnvironment service for determining the current environment, which avoids the need to get the configuration setting manually. The IWebHostEnvironment service defines the property and methods shown in table 15.3. The methods are extension methods that are defined in the Microsoft.Extensions.Hosting namespace.

Table 15.3 The IWebHostEnvironment extension methods

Name	Description
EnvironmentName	This property returns the current environment.
IsDevelopment()	This method returns true when the Development environment has been selected.
IsStaging()	This method returns true when the Staging environment has been selected.
IsProduction()	This method returns true when the Production environment has been selected.
IsEnvironment(env)	This method returns true when the environment specified by the argument has been selected.

If you need to access the environment when setting up services, then you can use the WebApplicationBuilder.Environment property. If you need to access the environment when configuring the pipeline, you can use the WebApplication.Environment property. If you need to access the environment within a middleware component or endpoint, then you can define a IWebHostEnvironment parameter. All three approaches are shown in listing 15.12.

Listing 15.12 Accessing the environment in the Program.cs file in the Platform folder

```
using Platform;

var builder = WebApplication.CreateBuilder(args);

var servicesConfig = builder.Configuration;
builder.Services.Configure<MessageOptions>(
    servicesConfig.GetSection("Location"));

var servicesEnv = builder.Environment;
// - use environment to set up services

var app = builder.Build();

var pipelineConfig = app.Configuration;
// - use configuration settings to set up pipeline

var pipelineEnv = app.Environment;
```

```
// - use envirionment to set up pipeline

app.UseMiddleware<LocationMiddleware>();

app.MapGet("config", async (HttpContext context,
        IConfiguration config, IWebHostEnvironment env) => {
    string? defaultDebug = config["Logging:LogLevel:Default"];
    await context.Response
        .WriteAsync($"The config setting is: {defaultDebug}");
    await context.Response
        .WriteAsync($"\nThe env setting is: {env.EnvironmentName}");
});

app.MapGet("/", async context => {
    await context.Response.WriteAsync("Hello World!");
});

app.Run();
```

Restart ASP.NET Core and use a browser to request http://localhost:5000/config, which produces the output shown in figure 15.8.

15.2.7 Storing user secrets

During development, it is often necessary to use sensitive data to work with the services that an application depends on. This data can include API keys, database connection passwords, or default administration accounts, and it is used both to access services and to reinitialize them to test application changes with a fresh database or user configuration.

If the sensitive data is included in the C# classes or JSON configuration files, it will be checked into the source code version control repository and become visible to all developers and to anyone else who can see the code—which may mean visible to the world for projects that have open repositories or repositories that are poorly secured.

The user secrets service allows sensitive data to be stored in a file that isn't part of the project and won't be checked into version control, allowing each developer to have sensitive data that won't be accidentally exposed through a version control check-in.

STORING USER SECRETS

The first step is to prepare the file that will be used to store sensitive data. Run the command shown in listing 15.13 in the `Platform` folder.

Listing 15.13 Initializing user secrets

```
dotnet user-secrets init
```

This command adds an element to the `Platform.csproj` project file that contains a unique ID for the project that will be associated with the secrets on each developer machine. Next, run the commands shown in listing 15.14 in the `Platform` folder.

Listing 15.14 Storing a user secret

```
dotnet user-secrets set "WebService:Id" "MyAccount"
dotnet user-secrets set "WebService:Key" "MySecret123$"
```

Each secret has a key and a value, and related secrets can be grouped together by using a common prefix, followed by a colon (the `:` character), followed by the secret name. The commands in listing 15.14 create related `Id` and `Key` secrets that have the `WebService` prefix.

After each command, you will see a message confirming that a secret has been added to the secret store. To check the secrets for the project, use the command prompt to run the command shown in listing 15.15 in the `Platform` folder.

Listing 15.15 Listing the user secrets

```
dotnet user-secrets list
```

This command produces the following output:

```
WebService:Key = MySecret123$
WebService:Id = MyAccount
```

Behind the scenes, a JSON file has been created in the `%APPDATA%\Microsoft\UserSecrets` folder (or the `~/.microsoft/usersecrets` folder for Linux) to store the secrets. Each project has its own folder (whose name corresponds to the unique ID created by the `init` command in listing 15.13).

> **TIP** If you are using Visual Studio, you can create and edit the JSON file directly by right-clicking the project in the Solution Explorer and selecting Manage User Secrets from the pop-up menu.

READING USER SECRETS

User secrets are merged with the normal configuration settings and accessed in the same way. In listing 15.16, I have added a statement that displays the secrets to the middleware component that handles the `/config` URL.

Listing 15.16 Using user secrets in the Program.cs file in the Platform folder

```
using Platform;

var builder = WebApplication.CreateBuilder(args);

var servicesConfig = builder.Configuration;
builder.Services.Configure<MessageOptions>(
    servicesConfig.GetSection("Location"));

var servicesEnv = builder.Environment;
// - use environment to set up services

var app = builder.Build();

var pipelineConfig = app.Configuration;
```

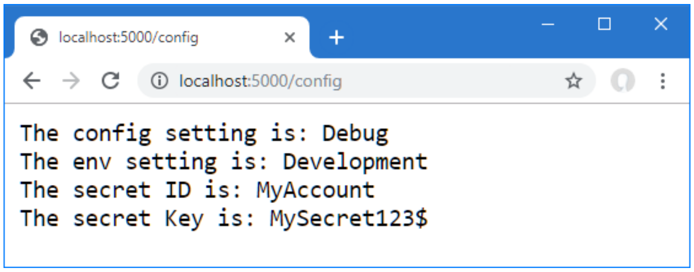
```
// - use configuration settings to set up pipeline

var pipelineEnv = app.Environment;
// - use envirionment to set up pipeline

app.UseMiddleware<LocationMiddleware>();

app.MapGet("config", async (HttpContext context,
        IConfiguration config, IWebHostEnvironment env) => {
    string? defaultDebug = config["Logging:LogLevel:Default"];
    await context.Response
        .WriteAsync($"The config setting is: {defaultDebug}");
    await context.Response
        .WriteAsync($"\nThe env setting is: {env.EnvironmentName}");
    string? wsID = config["WebService:Id"];
    string? wsKey = config["WebService:Key"];
    await context.Response.WriteAsync($"\nThe secret ID is: {wsID}");
    await context.Response.WriteAsync($"\nThe secret Key is: {wsKey}");
});

app.MapGet("/", async context => {
    await context.Response.WriteAsync("Hello World!");
});

app.Run();
```

User secrets are loaded only when the application is set to the `Development` environment. Edit the `launchSettings.json` file to change the environment to `Development`, as shown in listing 15.17.

Listing 15.17 Changing the launchSettings.json file in the Platform/Properties folder

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:5000",
      "sslPort": 0
    }
  },
  "profiles": {
    "Platform": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
            "ASPNETCORE_ENVIRONMENT": "Development"
          }
        }
      }
    }
```

Save the changes, restart the ASP.NET Core runtime using the `dotnet run` command, and request the http://localhost:5000/config URL to see the user secrets, as shown in figure 15.9.



```
The config setting is: Debug
The env setting is: Development
The secret ID is: MyAccount
The secret Key is: MySecret123$
```

Figure 15.9 Displaying user secrets

15.3 Using the logging service

ASP.NET Core provides a logging service that can be used to record messages that describe the state of the application to track errors, monitor performance, and help diagnose problems.

Log messages are sent to logging providers, which are responsible for forwarding messages to where they can be seen, stored, and processed. There are built-in providers for basic logging, and there is a range of third-party providers available for feeding messages into logging frameworks that allow messages to be collated and analyzed.

Three of the built-in providers are enabled by default: the console provider, the debug provider, and the `EventSource` provider. The debug provider forwards messages so they can be processed through the `System.Diagnostics.Debug` class, and the `EventSource` provider forwards messages for event tracing tools, such as `PerfView` (https://github.com/Microsoft/perfview). I use the console provider in this chapter because it is simple and doesn't require any additional configuration to display logging messages.

TIP You can see the list of providers available and instructions for enabling them at https://docs.microsoft.com/en-us/aspnet/core/fundamentals/logging.

15.3.1 Generating logging messages

To prepare for this section, listing 15.18 reconfigures the application to remove the services, middleware, and endpoints from the previous section.

Listing 15.18 Configuring the application in the Program.cs file in the Platform folder

```
using Platform;

var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.MapGet("population/{city?}", Population.Endpoint);

app.Run();
```

Logging messages are generated using the unbounded `ILogger<>` service, as shown in listing 15.19.

Listing 15.19 Generating logging messages in the Population.cs file in the Platform folder

```
namespace Platform {
    public class Population {

        public static async Task Endpoint(HttpContext context,
                ILogger<Population> logger) {
            logger.LogDebug("Started processing for {path}",
                context.Request.Path);
            string city = context.Request.RouteValues["city"]
                as string ?? "london";
            int? pop = null;
            switch (city.ToLower()) {
                case "london":
                    pop = 8_136_000;
                    break;
                case "paris":
                    pop = 2_141_000;
                    break;
                case "monaco":
                    pop = 39_000;
                    break;
            }
            if (pop.HasValue) {
                await context.Response
                    .WriteAsync($"City: {city}, Population: {pop}");
            } else {
                context.Response.StatusCode
                    = StatusCodes.Status404NotFound;
            }
            logger.LogDebug("Finished processing for {path}",
                context.Request.Path);
        }
    }
}
```

The logging service groups log messages together based on the category assigned to messages. Log messages are written using the `ILogger<T>` interface, where the generic parameter `T` is used to specify the category. The convention is to use the type of the class that generates the messages as the category type, which is why listing 15.19 declares a dependency on the service using `Population` for the type argument, like this:

```
...
public static async Task Endpoint(HttpContext context,
    ILogger<Population> logger) {
...
```

This ensures that log messages generated by the `Endpoint` method will be assigned the category `Population`. Log messages are created using the extension methods shown in table 15.4.

Table 15.4 The ILogger<T> extension methods

Name	Description
`LogTrace`	This method generates a `Trace`-level message, used for low-level debugging `during` development.
`LogDebug`	This method generates a `Debug`-level message, used for low-level debugging during development or production problem resolution.
`LogInformation`	This method generates an `Information`-level message, used to provide information about the general state of the application.
`LogWarning`	This method generates a `Warning`-level message, used to record unexpected, `but` minor, problems that are unlikely to disrupt the application.
`LogError`	This method generates an `Error`-level message, used to record exceptions or errors that are not handled by the application.
`LogCritical`	This method generates a `Critical`-level message, used to record serious failures.

Log messages are assigned a level that reflects their importance and detail. The levels range from `Trace`, for detailed diagnostics, to `Critical`, for the most important information that requires an immediate response. There are overloaded versions of each method that allow log messages to be generated using strings or exceptions. In listing 15.19, I used the `LogDebug` method to generate logging messages when a request is handled.

```
...
logger.LogDebug("Started processing for {path}", context.Request.Path);
...
```

The result is log messages at the `Debug` level that are generated when the response is started and completed. To see the log messages, restart ASP.NET Core and use a browser to request http://localhost:5000/population. Look at the console output, and you will see the log messages in the output from ASP.NET Core, like this:

```
Building...
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5000
```

```
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Platform
dbug: Platform.Population[0]
      Started processing for /population
dbug: Platform.Population[0]
      Finished processing for /population
```

LOGGING MESSAGES IN THE PROGRAM.CS FILE

The `Logger<>` service is useful for logging in classes but isn't suited to logging in the `Program.cs` file, where top-level statements are used to configure the application. The simplest approach is to use the `ILogger` returned by the `Logger` property defined by the `WebApplication` class, as shown in listing 15.20.

Listing 15.20 Logging in the Program.cs file in the Platform folder

```
using Platform;

var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.Logger.LogDebug("Pipeline configuration starting");

app.MapGet("population/{city?}", Population.Endpoint);

app.Logger.LogDebug("Pipeline configuration complete");

app.Run();
```

The `ILogger` interface defines all the methods described in table 15.4. Start ASP.NET Core, and you will see the logging messages in the startup output, like this:

```
Building...
dbug: Platform[0]
      Pipeline configuration starting
dbug: Platform[0]
      Pipeline configuration complete
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5000
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Platform
```

The category for logging messages generated using the `ILogger` provided by the `WebApplication` class is the name of the application, which is `Platform` for this example. If you want to generate log messages with a different category, which can be useful in lambda functions, for example, then you can use the `ILoggerFactory` interface, which is available as a service, and call the `CreateLogger` method to obtain an `ILogger` for a specified category, as shown in listing 15.21.

Listing 15.21 Creating a logger in the Program.cs file in the Platform folder

```
using Platform;

var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

var logger = app.Services
       .GetRequiredService<ILoggerFactory>().CreateLogger("Pipeline");

logger.LogDebug("Pipeline configuration starting");

app.MapGet("population/{city?}", Population.Endpoint);

logger.LogDebug("Pipeline configuration complete");

app.Run();
```

Restart ASP.NET Core, and you will see the following messages in the output produced as the application starts:

```
Building...
dbug: Pipeline[0]
      Pipeline configuration starting
dbug: Pipeline[0]
      Pipeline configuration complete
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5000
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Platform
```

15.3.2 Logging messages with attributes

An alternative approach to generating log messages is to use the `LoggerMessage` attribute, as shown in listing 15.22.

Listing 15.22 Using the attribute in the Population.cs file in the Platform folder

```
namespace Platform {
    public partial class Population {

        public static async Task Endpoint(HttpContext context,
                ILogger<Population> logger) {
            //logger.LogDebug("Started processing for {path}",
            //    context.Request.Path);
            StartingResponse(logger, context.Request.Path);
            string city = context.Request.RouteValues["city"]
                as string ?? "london";
            int? pop = null;
            switch (city.ToLower()) {
                case "london":
                    pop = 8_136_000;
                    break;
                case "paris":
                    pop = 2_141_000;
                    break;
                case "monaco":
                    pop = 39_000;
                    break;
            }
            if (pop.HasValue) {
                await context.Response
                    .WriteAsync($"City: {city}, Population: {pop}");
            } else {
                context.Response.StatusCode
                    = StatusCodes.Status404NotFound;
            }
            logger.LogDebug("Finished processing for {path}",
                context.Request.Path);
        }

        [LoggerMessage(0, LogLevel.Debug, "Starting response for {path}")]
        public static partial void StartingResponse(ILogger logger,
            string path);
    }
}
```

The `LoggerMessage` attribute is applied to `partial` methods, which must be defined in `partial` classes. When the application is compiled, the attribute generates the implementation for the method to which it is applied, resulting in logging, which Microsoft says offers better performance than the other techniques described in this section. Full details of how this feature works can be found at https://docs.microsoft.com/en-us/dotnet/core/extensions/logger-message-generator.

NOTE I do not doubt Microsoft's assertion that using the `LoggerMessage` attribute is faster, but I doubt it matters for most projects. Use the attribute if you find this approach easier to understand and maintain, but don't rush to adopt it just for the sake of a performance gain unless you have an application that doesn't meet its performance goals and you are sure that logging performance is contributing to the problem. I am confident that this will never be the case for most projects because of the nature of most web applications, but please get in touch if you find yourself in this position because I am always willing to have my assumptions proven wrong.

Start ASP.NET Core and use a browser to request http://localhost:5000/population, and the output will include the following log messages:

```
dbug: Platform.Population[0]
      Starting response for /population
dbug: Platform.Population[0]
      Finished processing for /population
```

15.3.3 Configuring minimum logging levels

Earlier in this chapter, I showed you the default contents of the `appsettings.json` and `appsettings.Development.json` files and explained how they are merged to create the application's configuration settings. The settings in the JSON file are used to configure the logging service, which ASP.NET Core provides to record messages about the state of the application.

The `Logging:LogLevel` section of the `appsettings.json` file is used to set the minimum level for logging messages. Log messages that are below the minimum level are discarded. The `appsettings.json` file contains the following levels:

```
...
"Default": "Information",
"Microsoft.AspNetCore": "Warning"
...
```

The category for the log messages—which is set using the generic type argument or using a string—is used to select a minimum filter level.

For the log messages generated by the `Population` class, for example, the category will be `Platform.Population`, which means that they can be matched directly by adding a `Platform.Population` entry to the `appsettings.json` file or indirectly by specifying just the `Platform` namespace. Any category for which there is no minimum log level is matched by the `Default` entry, which is set to `Information`.

It is common to increase the detail of the log messages displayed during development, which is why the levels in the `appsettings.Development.json` file specify more detailed logging levels, like this:

```
...
"Default": "Debug",
"System": "Information",
"Microsoft": "Information"
...
```

When the application is configured for the `Development` environment, the default logging level is `Debug`. The levels for the `System` and `Microsoft` categories are set to `Information`, which affects the logging messages generated by ASP.NET Core and the other packages and frameworks provided by Microsoft.

You can tailor the logging levels to focus the log on those parts of the application that are of interest by setting a level to `Trace`, `Debug`, `Information`, `Warning`, `Error`, or `Critical`. Logging messages can be disabled for a category using the `None` value.

Listing 15.23 sets the level to `Debug` for the `Microsoft.AspNetCore` setting, which will increase the default level of detail and will have the effect of displaying debug-level messages generated by ASP.NET Core.

TIP If you are using Visual Studio, you may have to expand the `appsettings.json` item in the Solution Explorer to see the `appsettings.Development.json` file.

Listing 15.23 Configuring the appsettings.Development.json file in the Platform folder

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information",
      "Microsoft.AspNetCore": "Debug"
    }
  }
}
```

Restart ASP.NET Core and request the http://localhost:5000/population URL, and you will see a series of messages from the different ASP.NET Core components. You can reduce the detail by being more specific about the namespace for which messages are required, as shown in listing 15.24.

Listing 15.24 Configuring the appsettings.Development.json file in the Platform folder

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information",
      "Microsoft.AspNetCore": "Warning",
      "Microsoft.AspNetCore.Routing": "Debug"
    }
  }
}
```

The changes return the `Microsoft.AspNetCore` category to `Warning` and set the `Microsoft.AspNetCore.Routing` category to `Debug`, which increases the detail level for logging messages by the components responsible for routing. Restart ASP.NET Core and request http://localhost:5000/population again, and you will see fewer messages overall, but still see those that report how the request was matched to a route:

```
...
dbug: Microsoft.AspNetCore.Routing.Matching.DfaMatcher[1001]
      1 candidate(s) found for the request path '/population'
dbug: Microsoft.AspNetCore.Routing.Matching.DfaMatcher[1005]
      Endpoint 'HTTP: GET population/{city?} =>
        Endpoint' with route pattern 'population/{city?}' is valid
        for the request path '/population'
dbug: Microsoft.AspNetCore.Routing.EndpointRoutingMiddleware[1]
      Request matched endpoint 'HTTP: GET population/{city?} => Endpoint'
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[0]
      Executing endpoint 'HTTP: GET population/{city?} => Endpoint'
dbug: Platform.Population[0]
      Starting response for /population
dbug: Platform.Population[0]
      Finished processing for /population
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[1]
      Executed endpoint 'HTTP: GET population/{city?} => Endpoint'
...
```

If you are having trouble figuring out a routing scheme, then these messages can be helpful in figuring out what the application is doing with requests.

15.3.4 Logging HTTP requests and responses

ASP.NET Core includes built-in middleware for generating log messages that describe the HTTP requests received by an application and the responses it produces. Listing 15.25 adds the HTTP logging middleware to the request pipeline.

Listing 15.25 Adding logging middleware in the Program.cs file in the Platform folder

```
using Platform;

var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.UseHttpLogging();

//var logger = app.Services
//   .GetRequiredService<ILoggerFactory>().CreateLogger("Pipeline");

//logger.LogDebug("Pipeline configuration starting");

app.MapGet("population/{city?}", Population.Endpoint);

//logger.LogDebug("Pipeline configuration complete");

app.Run();
```

The `UseHttpLogging` method adds a middleware component that generates logging messages that describe the HTTP requests and responses. These log messages are generated with the `Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware` category and the `Information` severity, which I have enabled in listing 15.26.

Listing 15.26 Logging in the appsettings.Development.json file in the Platform folder

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information",
      "Microsoft.AspNetCore": "Warning",
      "Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware":
          "Information"
    }
  }
}
```

Restart ASP.NET Core and request http://localhost:5000/population, and you will see logging messages that describe the HTTP request sent by the browser and the response the application produces, similar to the following:

```
...
info: Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware[1]
      Request:
      Protocol: HTTP/1.1
      Method: GET
      Scheme: http
      PathBase:
      Path: /population
      Accept: text/html,application/xhtml+xml,application/xml;q=0.9,
          image/avif,image/webp,image/apng,*/*;q=0.8,
          application/signed-exchange;v=b3;q=0.9
      Connection: keep-alive
      Host: localhost:5000
      User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
          AppleWebKit/537.36 (KHTML, like Gecko)
          Chrome/94.0.4606.71 Safari/537.36
      Accept-Encoding: gzip, deflate, br
      Accept-Language: en-GB,en-US;q=0.9,en;q=0.8
      Cache-Control: [Redacted]
      Cookie: [Redacted]
      Upgrade-Insecure-Requests: [Redacted]
      sec-ch-ua: [Redacted]
      sec-ch-ua-mobile: [Redacted]
      sec-ch-ua-platform: [Redacted]
      Sec-Fetch-Site: [Redacted]
      Sec-Fetch-Mode: [Redacted]
      Sec-Fetch-User: [Redacted]
      Sec-Fetch-Dest: [Redacted]
dbug: Platform.Population[0]
      Starting response for /population
dbug: Platform.Population[0]
      Finished processing for /population
info: Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware[2]
      Response:
      StatusCode: 200
      Date: [Redacted]
      Server: [Redacted]
      Transfer-Encoding: chunked
...
```

The details of the HTTP request and response logging messages can be configured using the `AddHttpLogging` method, as shown in listing 15.27.

Listing 15.27 HTTP logging messages in the Program.cs file in the Platform folder

```
using Platform;
using Microsoft.AspNetCore.HttpLogging;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddHttpLogging(opts => {
    opts.LoggingFields = HttpLoggingFields.RequestMethod
        | HttpLoggingFields.RequestPath
        | HttpLoggingFields.ResponseStatusCode;
});
```

```
var app = builder.Build();

app.UseHttpLogging();

app.MapGet("population/{city?}", Population.Endpoint);

app.Run();
```

This method selects the fields and headers that are included in the logging message. The configuration in listing 15.27 selects the method and path from the HTTP request and the status code from the response. See https://docs.microsoft.com/en-us/aspnet/core/fundamentals/http-logging for the complete set of configuration options for HTTP logging.

Restart ASP.NET Core and request http://localhost:5000/population, and you will see the selected details in the output:

```
...
info: Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware[1]
      Request:
      Method: GET
      PathBase:
      Path: /population
dbug: Platform.Population[0]
      Starting response for /population
dbug: Platform.Population[0]
      Finished processing for /population
info: Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware[2]
      Response:
      StatusCode: 200
...
```

NOTE ASP.NET Core also provides middleware that will generate log messages in the W3C format. See https://docs.microsoft.com/en-us/aspnet/core/fundamentals/w3c-logger for details.

15.4 Using static content and client-side packages

Most web applications rely on a mix of dynamically generated and static content. The dynamic content is generated by the application based on the user's identity and actions, such as the contents of a shopping cart or the detail of a specific product and is generated fresh for each request. I describe the different ways that dynamic content can be created using ASP.NET Core in part 3.

Static content doesn't change and is used to provide images, CSS stylesheets, JavaScript files, and anything else on which the application relies but which doesn't have to be generated for every request. The conventional location for static content in an ASP.NET Core project is the `wwwroot` folder.

To prepare static content to use in the examples for this section, create the `Platform/wwwroot` folder and add to it a file called `static.html`, with the content shown in listing 15.28. You can create the file with the HTML Page template if you are using Visual Studio.

Listing 15.28 The contents of the static.html file in the wwwroot folder

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Static Content</title>
</head>
<body>
    <h3>This is static content</h3>
</body>
</html>
```

The file contains a basic HTML document with just the basic elements required to display a message in the browser.

15.4.1 Adding the static content middleware

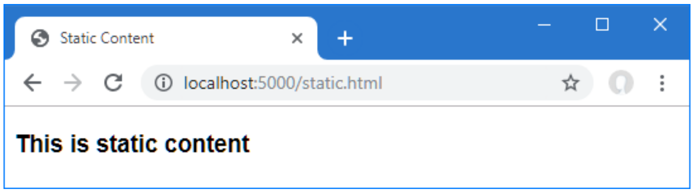
ASP.NET Core provides a middleware component that handles requests for static content, which is added to the request pipeline in listing 15.29.

Listing 15.29 Adding middleware in the Program.cs file in the Platform folder

```
using Platform;
using Microsoft.AspNetCore.HttpLogging;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddHttpLogging(opts => {
    opts.LoggingFields = HttpLoggingFields.RequestMethod
        | HttpLoggingFields.RequestPath
        | HttpLoggingFields.ResponseStatusCode;
});

var app = builder.Build();

app.UseHttpLogging();

app.UseStaticFiles();

app.MapGet("population/{city?}", Population.Endpoint);

app.Run();
```

The `UseStaticFiles` extension method adds the static file middleware to the request pipeline. This middleware responds to requests that correspond to the names of disk files and passes on all other requests to the next component in the pipeline. This middleware is usually added close to the start of the request pipeline so that other components don't handle requests that are for static files.

Restart ASP.NET Core and navigate to http://localhost:5000/static.html. The static file middleware will receive the request and respond with the contents of the `static.html` file in the `wwwroot` folder, as shown in figure 15.10.



This is static content

Figure 15.10 Serving static content

The middleware component returns the content of the requested file and sets the response headers, such as `Content-Type` and `Content-Length`, that describe the content to the browser.

Changing the default options for the static content middleware

When the `UseStaticFiles` method is invoked without arguments, the middleware will use the `wwwroot` folder to locate files that match the path of the requested URL.

This behavior can be adjusted by passing a `StaticFileOptions` object to the `UseStaticFiles` method. Table 15.5 describes the properties defined by the `StaticFileOptions` class.

Table 15.5 The properties defined by the StaticFileOptions class

Name	Description
`ContentTypeProvider`	This property is used to get or set the `IContentTypeProvider` object that is responsible for producing the MIME type for a file. The default implementation of the interface uses the file extension to determine the content type and supports the most common file types.
`DefaultContentType`	This property is used to set the default content type if the `IContentTypeProvider` cannot determine the type of the file.
`FileProvider`	This property is used to locate the content for requests, as shown in the listing below.
`OnPrepareResponse`	This property can be used to register an action that will be invoked before the static content response is generated.
`RequestPath`	This property is used to specify the URL path that the middleware will respond to, as shown in the following listing.
`ServeUnknownFileTypes`	By default, the static content middleware will not serve files whose content type cannot be determined by the `IContentTypeProvider`. This behavior is changed by setting this property to `true`.

The `FileProvider` and `RequestPath` properties are the most commonly used. The `FileProvider` property is used to select a different location for static content, and the `RequestPath` property is used to specify a URL prefix that denotes requests for static context. Listing 15.30 uses both properties to configure the static file middleware.

TIP There is also a version of the UseStaticFiles method that accepts a single string argument, which is used to set the RequestPath configuration property. This is a convenient way of adding support for URLs without needing to create an options object.

Listing 15.30 Configuring the static files in the Program.cs file in the Platform folder

```
using Platform;
using Microsoft.AspNetCore.HttpLogging;
using Microsoft.Extensions.FileProviders;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddHttpLogging(opts => {
    opts.LoggingFields = HttpLoggingFields.RequestMethod
        | HttpLoggingFields.RequestPath
        | HttpLoggingFields.ResponseStatusCode;
});

var app = builder.Build();

app.UseHttpLogging();

app.UseStaticFiles();

var env = app.Environment;
app.UseStaticFiles(new StaticFileOptions {
    FileProvider = new
        PhysicalFileProvider($"{env.ContentRootPath}/staticfiles"),
    RequestPath = "/files"
});

app.MapGet("population/{city?}", Population.Endpoint);

app.Run();
```

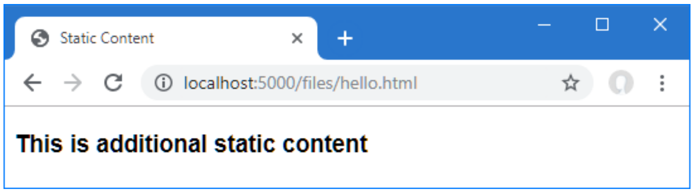
Multiple instances of the middleware component can be added to the pipeline, each of which handles a separate mapping between URLs and file locations. In the listing, a second instance of the static files middleware is added to the request pipeline so that requests for URLs that start with /files will be handled using files from a folder named staticfiles. Reading files from the folder is done with an instance of the Physical-FileProvider class, which is responsible for reading disk files. The PhysicalFile-Provider class requires an absolute path to work with, which I based on the value of the ContentRootPath property defined by the IWebHostEnvironment interface, which is the same interface used to determine whether the application is running in the Development or Production environment.

To provide content for the new middleware component to use, create the Platform/staticfiles folder and add to it an HTML file named hello.html with the content shown in listing 15.31.

Listing 15.31 The contents of the hello.html file in the Platform/staticfiles folder

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Static Content</title>
</head>
<body>
    <h3>This is additional static content</h3>
</body>
</html>
```

Restart ASP.NET Core and use the browser to request the http://localhost:5000/files/hello.html URL. Requests for URLs that begin with `/files` and that correspond to files in the `staticfiles` folder are handled by the new middleware, as shown in figure 15.11.



This is additional static content

Figure 15.11 Configuring the static files middleware

15.4.2 *Using client-side packages*

Most web applications rely on client-side packages to support the content they generate, using CSS frameworks to style content or JavaScript packages to create rich functionality in the browser. Microsoft provides the Library Manager tool, known as LibMan, for downloading and managing client-side packages.

PREPARING THE PROJECT FOR CLIENT-SIDE PACKAGES

Use the command prompt to run the commands shown in listing 15.32, which remove any existing LibMan package and install the version required by this chapter as a global .NET Core tool.

Listing 15.32 Installing LibMan

```
dotnet tool uninstall --global Microsoft.Web.LibraryManager.Cli
dotnet tool install --global Microsoft.Web.LibraryManager.Cli
    --version 2.1.175
```

The next step is to create the LibMan configuration file, which specifies the repository that will be used to get client-side packages and the directory into which packages will be downloaded. Open a PowerShell command prompt and run the command shown in listing 15.33 in the `Platform` folder.

Listing 15.33 Initializing LibMan

```
libman init -p cdnjs
```

The -p argument specifies the provider that will get packages. I have used cdnjs, which selects cdnjs.com. The other option is unpkg, which selects unpkg.com. If you don't have existing experience with package repositories, then you should start with the cdnjs option.

The command in listing 15.33 creates a file named libman.json in the Platform folder; the file contains the following settings:

```
...
{
  "version": "1.0",
  "defaultProvider": "cdnjs",
  "libraries": []
}
...
```

If you are using Visual Studio, you can create and edit the libman.json file directly by selecting Project > Manage Client-Side Libraries.

INSTALLING CLIENT-SIDE PACKAGES

Packages are installed from the command line. Run the command shown in listing 15.34 in the Platform folder to install the Bootstrap package.

Listing 15.34 Installing the Bootstrap package

```
libman install bootstrap@5.2.3 -d wwwroot/lib/bootstrap
```

The required version is separated from the package name by the @ character, and the -d argument is used to specify where the package will be installed. The wwwroot/lib folder is the conventional location for installing client-side packages in ASP.NET Core projects.

USING A CLIENT-SIDE PACKAGE

Once a client-side package has been installed, its files can be referenced by script or link HTML elements or by using the features provided by the higher-level ASP.NET Core features described in later chapters.

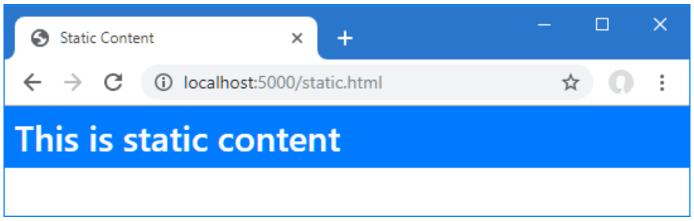
For simplicity in this chapter, listing 15.35 adds a link element to the static HTML file created earlier in this section.

Listing 15.35 Using a client package in the static.html file in the Platform/wwwroot folder

```
<!DOCTYPE html>
<html lang="en">
<head>
    <link rel="stylesheet" href="/lib/bootstrap/css/bootstrap.min.css" />
    <title>Static Content</title>
</head>
<body>
```

```
    <h3 class="p-2 bg-primary text-white">This is static content</h3>
</body>
</html>
```

Restart ASP.NET Core and request http://localhost:5000/static.html. When the browser receives and processes the contents of the `static.html` file, it will encounter the `link` element and send an HTTP request to the ASP.NET Core runtime for the `/lib/bootstrap/css/bootstrap.min.css` URL. The original static file middleware component will receive this request, determine that it corresponds to a file in the `wwwroot` folder, and return its contents, providing the browser with the Bootstrap CSS stylesheet. The Bootstrap styles are applied through the classes to which the `h3` element has been assigned, producing the result shown in figure 15.12.



This is static content

Figure 15.12 Using a client-side package

Summary

- The ASP.NET Core platform includes features for common tasks, must of which are presented as services.
- The configuration service provides access to the application configuration, which includes the contents of the `appsettings.json` file and environment variables.
- The configuration data is typically used with the options service to configure the services available through dependency injection.
- The user secrets feature is used to store sensitive data outside of the project folder, so they are not committed into a version control code repository.
- The logging service is used to generate log messages, with different severity levels and with options for sending the log messages to different handlers.
- ASP.NET Core includes middleware for serving static content and a tool for adding packages that will be delivered as static content to the project.